Rapport A3P

Projet Zuul/programmation d'un jeu en Java

Titre : Epopée éthylique à Dublin

Lien vers le site du jeu : http://www.esiee.fr/~houacinm/zuul/

Auteur : Houacine Mehdi, Groupe 5E

Table des matières :

Les éléments ci-dessus apparaîtront dans la version définitive du site web du jeu dont voici l'adresse : http://www.esiee.fr/~houacinm/zuul/ .

Tous les exercices ci-dessous ont été réalisés par Mehdi HOUACINE, ainsi que la rédaction de ce rapport et du site web de présentation du jeu.

I.A) Introduction :

La raison de la réalisation de ce jeu est un projet pédagogique proposé par l'école d'ingénieur ESIEE Paris à ses étudiants de première année (E1 premier cycle) entre le 1er février 2013 et le 13 mai 2013 pour apprendre la programmation en Java.

Vous verrez assez souvent sur le site ou dans ce rapport uniquement mon nom apparaître (Mehdi HOUACINE) car mon collègue de binôme (Louis Defloris) n'a pas participé au projet.

I.B) Thème du jeu :

Le thème validé par M.Bureau est :

« A la sortie d'un bar de Dublin, un homme ivre doit retrouver le chemin de sa maison. »

Ce thème résume donc le contexte du jeu, et le but fixé au joueur.

I.C) Résumé du scénario (complet) :

Dans ce jeu, vous débutez la partie dans un bar de Dublin, et vous contrôlez un homme qui recouvre peu à peu sa lucidité, et qui doit retournez chez lui de manière à ce que sa femme ne se rende pas compte qu'il soit sorti. A mesure que votre personnage sombre dans l'alcool, son mariage fait de même, et si sa femme le découvre dans cet état, ou si vous vous faites embarquer par la police la partie est perdue. Arriver chez vous sera une épreuve, arriver avec une bonne excuse en est une autre…puisque le fait que vous arriviez chez vous ne vous fera pas gagner, il faudra aussi persuader la femme du joueur que vous étiez dehors pour une bonne raison, raison que vous trouverez sur le chemin du retour. De plus, vous n'avez que 30 minutes de jeu pour rentrer chez vous (eh oui, vous êtes pressé) !

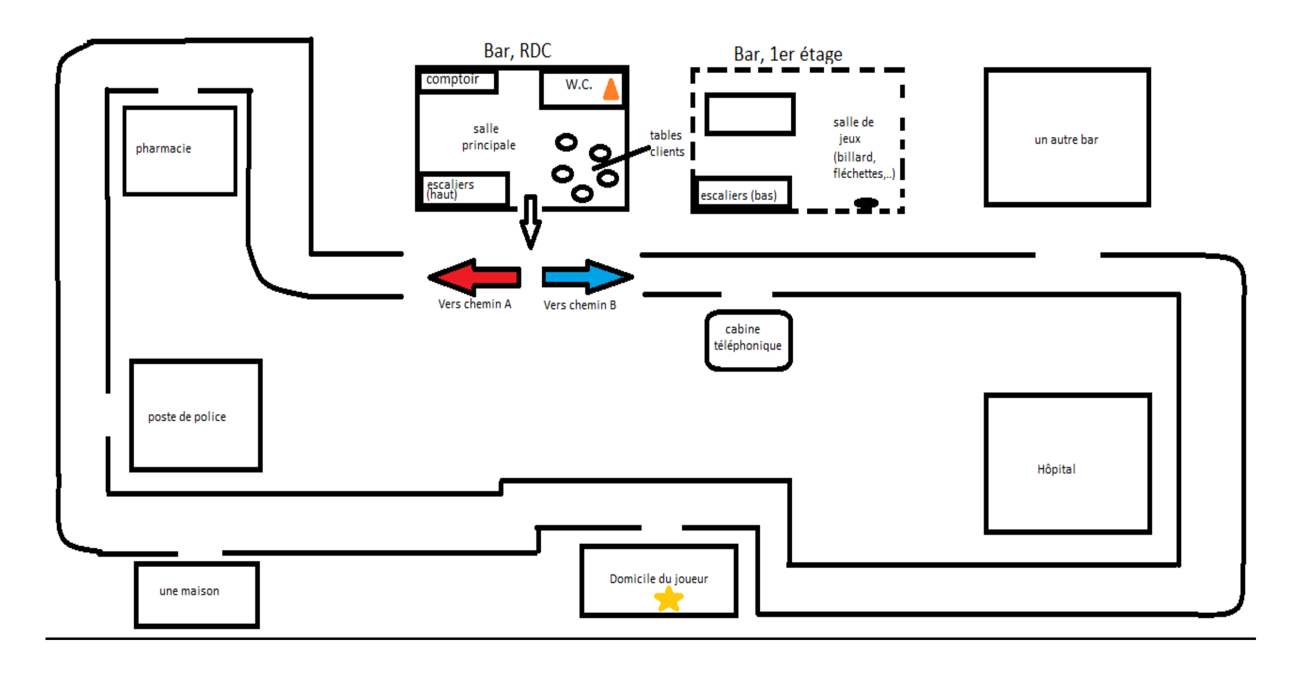
La partie débute dans les toilettes du bar, où le joueur se réveille après avoir perdu connaissance. Il se rappelle peu à peu les événements qui se sont déroulés avant son léger coma de 6h allongé sous le lavabo poisseux des toilettes d'un bar tout aussi crasseux des quartiers nord de Dublin.

Tout au long du jeu, le joueur devra surveiller sa lucidité, qui évoluera en fonction de certains choix. Certaines possibilités ou options de dialogues seront refusées au joueur si sa lucidité n'est pas suffisante. Saoul, le joueur aura en apparence l'air d'être perdu, agressif, bref personne ne voudra de vous. Lucide, vous bénéficierez d'un charisme qui pourra vous faire sortir de situations bien embarrassantes.

Une fois sorti du bar, le joueur aura une multitude de choix à faire, qui auront une influence sur la partie, et sur la fin (voir situations gagnantes / perdantes). Une fois qu'il aura fait le tour du bar (voir section scénario détaillé, partie bar) le joueur pourra sortir et aura un choix déterminant à faire pour la suite… à droite ou.. à gauche ?

En effet, nous avons choisi d'offrir aux joueurs la possibilité de choisir le chemin qu'il souhaite emprunter. Nous instaurons alors une situation de doute dans le jeu, puisque le joueur ne saura pas ce qui l'attend, et le nom de chaque chemin ne donne aucun véritable indice sur la difficulté ou la longueur du trajet. Aussi, une fois le joueur décidé, il ne pourra plus revenir en arrière. Evidemment, chaque chemin présente un contenu différent.

I.D) Plan (complet, avec indication de la partie "réduit" si exercice 7.3.3) :



Le joueur débute la partie dans les W.C. du bar (position signalée par le triangle orange). La progression se décompose en trois grandes phases: le bar, le chemin emprunté (une fois sorti du bar, le joueur doit choisir entre le chemin A et le chemin B et une fois décidé il ne doit plus revenir en arrière, aucun retour en arrière n'est possible en théorie) et enfin son domicile où sa femme l'attend (indiqué par l'étoile).

I.E) Scénario détaillé (complet, avec indication de la partie "réduit" si exercice 7.3.3) :

Cette partie, liée à la section scenario rentre dans les détails des zones du jeu. Attention, lire cette partie pourrait gâcher votre expérience du jeu, tout l'avancement y est raconté en détail.

LE BAR :

Dans le bar se trouvent : les toilettes où le joueur débute la partie, la salle principale du bar, le comptoir du barman, la table des clients, et à l'étage, il y a une salle de jeux.

La partie débute donc dans les toilettes du bar, où le joueur se réveille après avoir légèrement perdu connaissance. Il se réveille saoul. Les choix du joueur sont de s'approcher du lavabo (où il pourra s'humecter le visage et gagner un peu de lucidité), ou sortir des toilettes, ou entrer dans une des cabines de laquelle se dégage une aura particulière.

En sortant des toilettes, on se retrouve alors dans la salle principale du bar. Les options sont plus variées. Mais le joueur rencontre un problème, il n'a plus son portefeuille. C'est donc à vous d'explorer le bar pour le retrouver :

- Le joueur peut s'avancer vers la table des clients.
- Le joueur peut se rendre au comptoir où se trouve le barman, et acheter la boisson de son choix (alcool=perte de lucidité et restriction dans l'inventaire, café ou eau=gain de lucidité et plus de place dans l'inventaire). Le portefeuille est indispensable puisque les boissons sont payantes. Une option de dialogue vous permettra d'obtenir sa position (à l'étage d'après le barman).
- Enfin, le joueur peut sortir du bar.

LA BIFURCATION :

Le joueur doit maintenant choisir entre 2 chemins, tout retour étant impossible étant donné que vous êtes pressé, vous avez un temps limité pour rentrer chez vous.

LE CHEMIN A :

La pharmacie :

Le premier arrêt possible est la pharmacie. Vous y rencontrez un pharmacien de garde pour la nuit qui vous fait une offre : il vous donne gratuitement deux pilules provenant de son arrière-boutique. Vous pouvez les ingérer (commande manger…) pour découvrir leurs effets (l'une vous fait perdre connaissance et vous vous réveillez au début du jeu donc il faut vous dépêcher pour rattraper votre retard, l'autre vous fais aussi perdre connaissance mais vous vous réveillez dans la pharmacie dépouillé (saoul, sans argent, et votre inventaire est vide). Rien ne vous oblige cependant à les ingérer et vous pourrez même vous en servir plus tard.

Le commissariat :

Le deuxième arrêt possible est le commissariat. Si vous y entrez, vous pouvez soit perdre, soit trouver une bonne excuse à raconter à votre femme (voir situations gagnantes/perdantes).

La maison familière :

Le dernier arrêt est une maison qui semble très familière, mais qui perdra cet attrait si vous y entrez (voir situations gagnantes/perdantes).

LE CHEMIN B :

La cabine téléphonique :

Le premier arrêt est une cabine téléphonique qui comme la cabine des toilettes du premier bar, est une cabine de téléportation (imposée lors du projet). On peut le voir comme un piège.

L'autre bar :

Le deuxième arrêt est un second bar qui peut vous faire perdre si vous ne vous contrôlez pas (voir situation gagnantes/perdantes).

L'hôpital :

Le dernier arrêt est un hôpital dans lequel se trouve une infirmière qui peut vous aidez à trouver une excuse à raconter à votre femme, si vous remplissez certaines conditions…

LA MAISON DU JOUEUR :

Les chemins A et B se croisent à la maison du joueur. Ca y est, la fin approche. En vous rendant à l'étage vous devrez faire face à la femme du joueur. Si vous voulez gagner vous avez intérêt à pouvoir lui fournir une bonne raison de ne pas vous tu-.. jeter dehors.(Voir les situations gagnantes/perdantes).

I.F) Détail des lieux, items, personnages :

Les lieux sont détaillés dans le scénario/scénario détaillé du jeu ci-avant. La cabine des toilettes du premier bar et la cabine téléphonique du chemin B sont des transporterRooms (vous y entrez, et vous ressortez dans une salle sélectionnée aléatoirement).

Les items qu'il est possible de trouver dans le jeu sont :

- Des boissons : de l'eau et du café sont présents pour aider le joueur à retrouver sa lucidité et lui accorder des bonus (augmenter le poids maximum portable par le joueur), de l'alcool la lui fait perdre et apporte des malus (baisse du poids maximum portable).
- Un cookie magique, imposé pour le projet, qui augmente le poids max portable du joueur de 50%.
- Le portefeuille du joueur, qui contient de l'argent, utilisable pour acheter de l'eau, ou du café au comptoir du bar.
- Deux téléporteurs, imposés pour le projet, qui se chargent dans une salle, et qui permettent de s'y rendre instantanément quand on le souhaite.
- Une médaille du mérite délivrée par la police sous certaines conditions.
- Un certificat médical délivré par l'hôpital de la ville sous certaines conditions.
- Des pilules offertes par le pharmacien dans la pharmacie du chemin A.
- Une table de billard (aucune utilité pour le déroulement du jeu) à l'étage du premier bar.

Les personnages clef du jeu sont quant à eux le barman du premier bar qui vous indiquent l'emplacement de votre portefeuille, le pharmacien de la pharmacie du chemin A, l'infirmière de l'hôpital du chemin B qui peut vous délivré le certificat médical, le policier du commissariat du chemin A qui peut vous récompenser de la médaille du mérite, et enfin votre femme qui se trouve au domicile du joueur. La réussite de la partie dépend entièrement de ce dernier personnage.

I.G) Situations gagnantes et perdantes :

Nous détaillerons ces situations pièce par pièce.

De manière générale, il faut éviter d'être saoul avant de passer d'une salle à une autre pour éviter les mauvaises surprises, et se rappeler que le joueur que vous incarnez est pressé: si vous mettez plus de 30 minutes pour arriver chez vous, vous avez perdu.

LE BAR :

Dans le bar, rien ne peut nous faire perdre mais c'est ici que vous pouvez changer d'état (saoul ou lucide) en ingérant du café, de l'eau ou au contraire de l'alcool.

LE CHEMIN A :

La pharmacie :

Le pharmacien de garde vous propose des "pilules" gratuitement. Cette générosité cache un piège. Une de ces pilules vous fait retourner au début du jeu (attention le chrono n'a pas recommencé) tandis que l'autre vous fait perdre tous vos objets et vous fait perdre votre lucidité. Mais après tout rien ne vous oblige à les ingérer. Vous pouvez en particulier les garder pour la zone suivante.

Le commissariat :

Si vous êtes lucide, et que vous avez au moins une de ces pilules sur vous, vous pourrez dénoncer le pharmacien à un policier. Il récupèrera ces pilules et vous donnera une médaille pour vous récompenser. Elle vous servira à la fin du jeu.

Si vous n'êtes pas lucide et que vous avez au moins une de ces pilules sur vous le policier vous arrête et la partie est terminée.

Si vous n'êtes pas lucide, mais que vous n'avez pas de pilule sur vous, le policier vous demande de quitter les lieux simplement.

La maison familière :

Cette maison est un piège. Elle ressemble à la vôtre, mais ce n'est pas le cas.

Si vous êtes lucide, vous pourrez expliquer tranquillement la situation au propriétaire qui vous laissera partir.

Si vous n'êtes pas lucide, le propriétaire appelle la police et vous avez perdu.

LE CHEMIN B :

La cabine téléphonique :

C'est un piège en soi puisque c'est une TransporterRoom: elle vous emmènera dans n'importe quelle salle du jeu si vous essayez d'en sortir. (astuce : vous pouvez utiliser la commande retour...)

L'autre bar :

Si vous êtes lucide, vous décidez de quitter ce bar pour éviter de redevenir saoul.

Si vous êtes saoul, vous accourez au comptoir et retombez dans l'alcool. La partie est finie.

L'hôpital :

Si vous êtes lucide, vous pouvez expliquer votre situation à une infirmière qui accepte de vous faire un certificat médical. Vous en aurez besoin à la fin du jeu.

Si vous n'êtes pas lucide, l'infirmière refuse de vous aider. Vous perdez votre sang-froid et elle appelle la police qui vous arrête. La partie est terminée.

LA MAISON DU JOUEUR :

C'est la dernière ligne droite, votre femme se trouve à l'étage.

Si vous êtes saoul, elle le voit tout de suite. La partie est terminée.

Si vous êtes saoul et que vous avez une bouteille d'alcool ou les pilules du pharmacien sur vous (il existe un moyen d'aller à la pharmacie même en choisissant le chemin B, avec une TransporterRoom ou un téléporteur bien employé), vous perdez l'équilibre et elle(s) tombe(nt) de votre poche. Votre femme enrage, la partie est un échec total.

Si vous êtes lucide, mais que vous n'avez pas d'excuse à présenter, la partie est un échec, sauf si vous avez réussi à arriver ici en moins de 5 minutes (depuis le début du jeu dans les toilettes). Elle considèrera alors votre retard comme acceptable, et la partie est gagnée. Mais vous n'aurez certainement pas exploré le jeu en vous dépêchant.

Si vous êtes lucide, et que vous avez au moins une bonne excuse (médaille de la police, ou certificat de l'hôpital) votre femme accepte de vous laisser entrer dans la chambre. Vous gagnez donc la partie.

I.H) Eventuellement énigmes, mini-jeux, combats, etc :

Il n'y a pas d'énigmes au sens traditionnel, mais c'est un jeu où il est rare de gagner du premier coup. Il faudra donc apprendre de vos erreurs pour comprendre pourquoi vous perdez et comment surmonter les obstacles. Ce qui en soit peut se voir comme la définition d'une énigme.

Nous n'avons pas posé d'énigme au sens conventionnel dans le jeu car il s'agit d'un jeu court, où une énigme n'a pas sa place. Une fois que l'énigme a été résout une fois, elle perd tout son intérêt. Aussi, une énigme en ce sens n'aurait pas sa place dans un jeu avec ce contexte.

I.I) Commentaires (ce qui manque, reste à faire, ...) :

Globalement, bous avons été dans les temps pendant la quasi-totalité du projet mais si nous avions un peu de retard sur le calendrier fixé dans la liste officielle des exercices. Nous sommes donc parvenus au bout de cette liste.

Nous aurions toutefois voulu, si le temps nous le permettait, améliorer le scénario, créer un véritable « background » en ce qui concerne les personnages et le joueur pour donner un véritable intérêt à ce jeu qui en l'état n'est qu'un jeu d'exploration.

Nous aurions aussi préféré finir l'exercice 47 de la liste des exercices qui aurait été un vrai plus en matière de couplage, vous le comprendrez certainement en voyant l'état de la classe GameEngine, et finir les exercices optionnels de la classe Door qui aurait fourni une méthode plus « propre » à mon goût que le système de la trap door, et de la sauvegarde de jeu qui elle est fonctionnelle mais demande plus de travail.

Nous avions aussi imaginé aux débuts du projet, quand nous n'avions pas encore une idée globale de ce que serait le jeu, la possibilité de jouer au billard ou aux fléchettes à l'étage si le joueur était lucide, ou faire influer l'alcool sur les déplacements du joueur s'il se déplaçait un pas après l'autre et non zone par zone en utilisant un algorithme pour gérer la façon de marcher du joueur étant saoul (méthode de Monte Carlo par exemple), mais nos compétences et la direction qu'a pris le jeu ne nous le permettent pas.

Nous avions aussi imaginé au début un moyen de faire ressentir au joueur son état d'ébriété lors du déplacement d'une room à l'autre en allant dans une direction aléatoire autre que celle voulu (ex : le joueur a son attribut aLucide sur false, il tape go sud, mais c'est go nord qui est choisi aléatoirement). Mais nous pensons que cela aurait rendu le jeu trop difficile. Nous aurions peut-être pu faire quelque chose de plus simple mais qui perd son intérêt au bout de 2 minutes de jeu consisterait à faire interpréter au jeu la direction opposée (donc qui n'est plus aléatoire, ex : je tape go sud, la machine comprend go nord). Cela perd en effet son intérêt quand on a compris le système et cela embête le joueur plus qu'autre chose et ne l'incite plus à découvrir ce qu'il se serait passé en allant à tel endroit en étant saoul.

Enfin, mon dernier regret concerne la lucidité du joueur qui est réduite à un état binaire. Mon intention initiale était de créer une sorte de jauge de lucidité qui évoluerait avec le temps mais cela aurait été trop compliqué à mettre en place (la construire, l'afficher constamment pour ne pas surprendre trop brusquement le joueur, gérer son évolution (tend vers l'ébriété avec le temps qui passe), placer dans le jeu à plusieurs endroits de quoi trouver de l'eau ou du café pour faire baisser ou augmenter cette jauge, créer plus d'interactions, etc…)

II. Mode d'emploi :

Les commandes:

go : permet de se déplacer dans une direction (go nord, ou sud, est, ouest, entree, sortie, etage, descendre).

retour : permet de retourner dans la dernière salle visitée.

regarder : pour faire l'état des lieux et celui du joueur.

objets : pour consulter l'inventaire du joueur.

ide : rappelle les commandes de jeu sans indications.

prendre : pour prendre un objet et le mettre dans votre inventaire. Attention à son poids et son prix !

lacher : pour déposer un objet qui est dans votre inventaire.

manger : pour manger ou boire un objet consommable qui est dans votre inventaire. Certains objets modifient les caractéristiques du joueur (lucidité, poids max portable, etc).

charge : pour charger un téléporteur de votre inventaire. Il enregistre la salle courante pour vous y amener plus tard.

teleportation : pour vous téléporter là ou vous avez chargé le téléporteur de votre inventaire.

save nomDeFichier : pour l'instant, permet de sauvegarder le poids max portable du joueur.

load nomDeFichier : pour charger le fichier de sauvegarde nommé "nomDeFichier".

POUR JOUER :

Rendez-vous sur http://www.esiee.fr/~houacinm/zuul/jeu . Acceptez les conditions et l'applet s'ouvrira dans une nouvelle page.

III. Déclaration obligatoire anti-plagiat (préciser toutes les parties de code que vous n'avez pas écrites vous-même et citez la source, sauf les fichiers zuul-*.jar qui sont fournis évidemment) :

Ce n'est pas à préciser mais le jeu trouve ses fondations dans le projet « zuul-bad » et ses déclinaisons de Michael Kolling and David J. Barnes dont les noms apparaissent dans la documentation.

Des emprunts de code ont été effectués à :

- L.DUPONT (E1 groupe 5L projet Zuul 2012/2013) pour réaliser le téléporteur et les salles TransporteurRoom et pour de nombreux débugs.

- A.LE (E1 projet Zuul 2012/2013) pour réaliser la commande retour().

- Aymeric KOEPPEL, Sébastien HU, Frédéric NGUYEN, Justine BOUGNOL (étudiants à ESIEE Paris): projet Zuul 2011/2012 "Death Ception" pour le Timer.

Leurs noms apparaissent aussi dans la documentation pour les classes concernées.

IV.A) Réponses aux exercices (autres que I.) :

Exercice 7.1 : découverte de Zuul-bad :

While exploring the application (Zuul-bad), answer the following questions (and write the answers into the report):

What does this application do?

Le jeu Zuul-Bad permet de visiter une université. Il nous propose des directions à choisir selon le lieu : les directions cardinal nord, sud,...

What commands does the game accept?

Le jeu accepte les commandes « go+direction proposée », ou « quit », ou « help ».

What does each command do?

« Help » nous décrit la situation dans laquelle on est (« You are lost. You are alone. You wander around at the university. ». Il nous rappelle aussi les commandes.

« Quit » permet de mettre fin au jeu (« Thank you for playing. Good bye. »)

« Go » permet de jouer, il permet de se déplacer d'un endroit à l'autre en précisant la direction proposée (north, south, east, west).

How many rooms are in the scenario?

Il y a 5 endroits à visiter : l'entrée principale de l'université, un amphithéâtre, le bar du campus, un laboratoire informatique, et le bureau de l'administrateur informatique.

Exercice 7.1.1 : le thème de notre jeu :

Choisir un thème pour votre jeu.

Le thème validé par le professeur est :

A la sortie d'un bar de Dublin, un homme ivre doit retrouver le chemin de sa maison.

Notre jeu présentera l'aventure d'un homme ivre qui doit retourner chez lui. Des icônes faisant référence à l'alcool y seront présentes, mais il ne s'agit pas d'inciter le joueur à boire, puisque, si ces choix sont proposés ils seront punitifs et freineront au possible la progression du joueur.

Exercice 7.2 : Le rôle des classes dans Zuul-bad :

After you know what the whole application does, try to find out what each individual class does.
Write down for each class the purpose of the class.
You need to look at the source code to do this. Note that you might not (and need not) understand all of the source code. Often, reading through comments and looking at method headers is enough.

Le rôle de chaque classe est :

- La classe *CommandWords* recense les mots-clés autorisés et détermine si une commande de l'utilisateur est valide.

- La classe *Command* engendre une commande de l'utilisateur.

- La classe *Room* permet de créer les pièces du jeu, une pièce représente une scène.

- La classe *Parser* permet de lire le texte entré par l'utilisateur et le transforme en commande.

- La classe *Game* est la classe principale du jeu (méthode play).

Ces informations sont tirées de la documentation du jeu.

Ex 7.2.1 : présentation de la classe scanner :

Comprendre l'utilisation de la classe Scanner dans zuul-bad pour lire au clavier

D'après les informations à notre disposition :

« Comprendre l'utilisation de la classe Scanner dans *zuul-bad* pour lire au clavier :

- System.in désigne le clavier (comme System.out désigne l'écran)
- Il faut créer un objet Scanner en lui passant *le clavier* en paramètre :
  Scanner vScan = new Scanner(System.in);
- Pour pouvoir compiler, il faut indiquer au compilateur où se trouve la classe Scanner :
  import java.util.Scanner;
- Il faut créer une String pour contenir les caractères qui seront tapés au clavier :
  String vLigne;
- Ensuite, à chaque fois qu'on écrira l'instruction
  vLigne = vScan.nextLine();
  toute la ligne de caractères tapés au clavier sera stockée dans vLigne
- En attendant que des caractères soient tapés au clavier, le programme reste bloqué. Il est donc indispensable de prévenir l'utilisateur de ce qu'on attend de lui, en affichant quelque chose comme "Veuillez taper une commande : " ou plus simplement "> " »

C'est donc une classe qui nous permet de rentrer des informations au clavier que la machine pourra récupérer et réutiliser.

Exercice 7.3 le scenario :

Design your own game scenario. Do this away from the computer. Do not think about implementation, classes, or even programming in general. Just think about inventing an interesting game. This could be done with a group of people.
The game can be anything that has as its base structure a player moving through different locations.

Le scénario sera présent sur le site web du jeu.

http://www.esiee.fr/~houacinm/zuul/ (voir scenario ou scenario détaillé)

Exercice 7.3.2:Dessiner un plan du jeu :

Dessiner un plan du jeu faisant bien apparaître la géographie des lieux, le nom des lieux, ainsi que les passages possibles entre-eux; l'incorporer au Rapport (Ce peut être un plan dessiné à la main, puis scanné).

Le plan du jeu est sur le site web.

http://www.esiee.fr/~houacinm/zuul/ (voir plan du quartier).

Exercice 7.4 : la création des zones du jeu :

Nous avons modifier le code source préexistant de Zuul-bad dans un nouveau projet comme demandé. Nous avons modifié la méthode CreateRooms() dans la classe Game pour créer nos zones de jeu en remplaçant le nom des zones de Zuul-bad par les notres :
outside = new Room("outside the main entrance of the university");

Est devenu ainsi par exemple

Room toilettes ;

toilettes = new Room("dans les toilettes du bar");

avec le nom de la zone avant « =new Room » et un descriptif entre parenthèses. La room « toilettes » a été déclarée juste avant.

http://www.esiee.fr/~houacinm/zuul/ (voir plan du quartier avec chaque zone).

Exercice 7.5 : printLocationInfo :

La méthode a été créée dans le jeu, comme indiquée dans le livre. Elle affiche le descriptif de la zone courante lors d'un changement de salle.

Exercice 7.6 : la méthode getExit :

La méthode a été créée dans le jeu à partir du modèle du livre. On a donc:

```
public Room getExit(String direction)

   {   return exits.get(direction);    }
```

Exercice 7.7: méthode getExitString :

Modifications de printlocation info

De la même manière que précédemment, nous avons rédiger une méthode getExitString dans la classe Room qui s'occupe de générer les informations sur un lieu pour pouvoir ensuite l'appeler dans la méthode printLocationInfo.

Exercice 7.8:HashMap, setExit :

Nous avons importé l'outil HashMap comme détaillé dans le livre puis modifié la méthode setExit pour parer aux erreurs de compilations induites par les changements imposés dans le livre.

```
public void setExit(String direction, Room neighbor)

   {

       exits.put(direction, neighbor);

   }
```

On peut donc ajouter des sorties à la HashMap nommée exits. Une zone aura donc une sortie(« direction ») et une zone associée à cette sortie(la « neighbor » donc zone voisine).

Exercice 7.8.1 : ajouter un déplacement vertical :

Dans notre bar, nous avions dessiné l'intégralité des pièces sur un étage. Nous avons donc profité de cet exercice pour implanter une pièce en hauteur : la salle de jeux se trouve maintenant au premier étage du bar.
Rajouter cette nouvelle direction « haut », et par la même occasion « bas » fut simple grâce à l'importation de la classe HashMap.

bar.setExit("etage", jeux);

permet de poser la direction vers l'étage à la zone « bar » vers « la salle de jeux ».

Exercice 7.9 : Que fait la méthode keySet :

La méthode keySet permet de lister l'ensemble des clefs (ici les directions) du tableau associatif (ici de la HashMap).

Exemple : exits.keySet() ;

Renvoie l'ensemble des directions de la HashMap<String, Room> nommée exits.

Exercice 7.10.1 et 2 : Compléter/générer la javadoc :

Les commentaires sont à jour pour l'instant et la javadoc consultable sur le site

http://www.esiee.fr/~houacinm/zuul/ (cliquer sur l'icône de la javadoc).

Edit : Les progdoc/userdoc sont aussi consultables désormais. Chaque méthode possède un commentaire en en-tête qui la décrit ainsi que des compléments @param, @return.

Exercice 7.11 : getLongDescription :

Ajout de la méthode dans Room d'après le livre, et modification de printLocationInfo() apportée.

Edit : en fin de projet voici la méthode :

```
/**

    * Affiche une description détaillée de cette pièce.

    * @return     une description de la pièce, les objets et personnages qui s'y trouvent, avec les sorties.

    */

   public String getLongDescription()

   {  return "Vous êtes " + description + ".\n"+ aItemListRoom.getListDescription() +".\n"+getListCharacter()+"\n"+  getExitString()+"\n";    }
```

Exercice 7.12 et 13 : optionnels mais non compris.

Exercice 7.14 : look()

Ajout de la commande « regarder » dans CommandWord, de la méthode regarder() dans Game, et d'un else if dans Game. Elle donne des informations sur la salle courante.

Edit : en fin de projet la méthode est :

```
/**
    * Permet de faire l'etat des lieux.
    * @return     la description de la pièce où l'on se trouve et l'etat(attributs argent, poids et lucidite) du joueur.
    */
   private void look()
   {
gui.println(aPlayer.getPlayerCurrentRoom().getLongDescription()+
aPlayer.etatJoueur()+"\n");
   }
```

Exercice 7.15 : eat()

Même processus pour ajouter une commande eat() qui pour l'instant ne permet rien (elle affiche juste un texte).

Edit : en fin de projet, la méthode ne contente plus d'afficher simplement une phrase. Elle est essentielle à la progression. On peut manger des objets consommables de l'inventaire et ceux-ci ont un effet sur les attributs du joueur :

```
/**
  * Permet de manger quelque chose.
  * Certains objets modifient les attributs du joueur en bien ou en mal.
*/
   private void eat(Command command)
   {
      String itemString = command.getSecondWord();
      Item vItem = aPlayer.getInventaire().getItem(itemString);

      if(!command.hasSecondWord())//s'il n'y a pas de second mot
        {
           gui.println("Que voulez-vous manger ? \n");
        }
```

```
        else if(!aPlayer.containItemInventaire(itemString))//si l'objet n'est pas dans
l'inventaire
        {
           gui.println("Il faut que cet objet soit dans votre inventaire !");
        }
        else
        {
         gui.println("Vous mangez un(e) : " + itemString + "\n");
          if(vItem.getConsommableItem()==true)//si l'objet est consommable
          {
             if(itemString.equals("eau")||itemString.equals("cafe"))//si c'est du café ou
de l'eau
             {

aPlayer.setPoidsMax(aPlayer.getPoidsMax()+20*(aPlayer.getPoidsMax())/100);//l'
attribut augmente de 20%
               aPlayer.setLucidite(true);//le joueur devient lucide
               gui.println("Vous recuperez en lucidite, et votre charge d'inventaire
augmente de 20% !");
             }
             else if(itemString.equals("cookie"))//si l'objet est un cookie
             {
                aPlayer.setPoidsMax(aPlayer.getPoidsMax()+(
50*(aPlayer.getPoidsMax())/100) );//l'attribut augmente de 50%
               gui.println("Vous venez de manger le cookie magique! Votre charge
d'inventaire augmente de moitie !");
             }
             else if(itemString.equals("alcool"))//si booze
             {
                aPlayer.setPoidsMax(aPlayer.getPoidsMax()-
20*(aPlayer.getPoidsMax())/100);//l'attribut diminue de 20%
                aPlayer.setLucidite(false);//le joueur perd sa lucidite (=saoul)
             }
             else if(itemString.equals("cachet"))//si c'est la pilule cachet(=drogue)
             {
                aPlayer.setLucidite(false);//perd sa lucidite
```

```
            aPlayer.setArgent(0.0);//perd son argent
            aPlayer.getInventaire().clearItemList();//perd tous les objets de son
inventaire
//          restart();
            gui.println("Vous avez ingere le cachet.");
            gui.println("Si ce n'etait pas deja le cas, vous avez perdu votre lucidite.");
            gui.println("Vous avez perdu connaissance, a votre reveil vous avez perdu
tout votre argent, et tout les objets de votre inventaire.");
//          gui.println("Aussi, vous avez perdu connaissance et vous vous reveillez
dans les toilettes du bar !");
            gui.println("Vous y reflechirez a deux fois la prochaine fois que vous
ingererez une pilule donnee par un inconnu !");
            gui.println("S'il vous reste au moins une de ces pillules, vous pourriez
denoncer le pharmacien au commissariat du chemin ouest.\n");
          }
          else if(itemString.equals("comprime"))//si c'est la pilule comprime
          {
            restart();//reprend le jeu dans les toilettes sans perte d'attribut.
            gui.println("Vous avez perdu connaissance et vous vous reveillez dans les
toilettes du bar !");
            gui.println("Depechez-vous, l'heure tourne et vous etes revenu a votre
point de depart !");
            gui.println("Vous y reflechirez a deux fois la prochaine fois que vous
ingererez une pillule donnee par un inconnu !");
            gui.println("S'il vous reste au moins une de ces pillules, vous pourriez
denoncer le pharmacien au commissariat du chemin ouest.\n");
           }
        }
        else {gui.println("Soyez raisonnable, vous n'allez pas manger ça...");}
            aPlayer.getInventaire().removeItem(itemString);   //l'objet disparait de
l'inventaire une fois mange.
   }
}
```

Exercice 7.16 : showAll, showCommands :

Ajout des méthodes showAll et showcommands pour afficher la liste des commandes comme indiqué dans le livre.

Exercice 7.17 : ajouter une commande et implication sur Game :

Il faut changer la classe game pour ajouter les nouvelles commandes, comme nous l'avons fait pour eat() et look().

Edit : en fin de projet, il faut ajouter la nouvelle commande à la structure « switch »de GameEngine, ainsi que le mot commande correspondant dans la classe CommandWord.

Exercice 7.18 : getCommandList :

Changement de showall en getCommandList comme indiqué dans le livre et autres modifications.

7,18,1 : projet Zuul better :

Modifications apportées comme demandé. Le projet compile.

Exercice optionnel 7.18.2 : stringbuilder :

Objet mutable qui peut donc changer au cours du temps. On peut construire petit à petit une chaîne de caractère, avec la méthode append() qui ajoute des caractères à la suite des autres, et toString qui retourne la chaîne.

(Source : informations et méthodes provenant de la javadoc 7).

Exercice 7.18.3 : chercher des images :

Nous avons trouvé la quasi-totalité des images pour chaque zone du jeu, (trouvée par recherches dans google images, en attendant de trouver mieux. Il est très difficile de trouver des images satisfaisantes pour le jeu ET libres de droits d'utilisation).

Edit : en fin de projet, ces images ont été gardées par défaut et chaque Room a son image. Un problème persiste cependant pour la première salle où l'image n'apparaît pas toujours. Lors de certaines parties, l'image n'est pas trouvée par le programme qui n'affiche donc que le champ de texte. Mais pour chaque partie, cette image apparaît toujours lorsqu'on quitte la salle et qu'on y revient.

Exercice 7.18.4 :Décider du titre du jeu :

Le titre du jeu a été décidé dès le début, dès que notre thème a été validé. Il s'agit de « Epopée éthylique à Dublin », il est présent dans le titre de notre rapport de TP, dans le jeu (affichage de bienvenue), et sur le site web.

Exercice 7.18.5 optionnel: HashMap des Room :

Elle a été créée comme demandée. Il s'agit d'une HashMap‹String,Room› aToutesRooms composée d'une String qui est le nom de la Room, et de la Room en question. Elle permet en effet d'accéder très facilement à une Room dans GameEngine, ou dans une autre classe via accesseur et la méthode get() commune aux HahMap, exemple : aToutesRooms.get(« comptoir ») ; pour obtenir la Room comptoir.

Exercice 7.18.6 :Zuul with images :

Nous avons téléchargé et étudié le projet zuul-with-image, qui apporte au projet une interface graphique, donc une nouvelle manière de jouer (apparition d'images pour chaque Room, et zone d'entré de texte avec un certain design en bas de page du terminal grâce à l'agencement des panels).

Il apporte deux nouvelles classes, GameEngine et UserInterface qui permettent de s'adapter à la nouvelle interface graphique.

GameEngine permet d'apporter l'image puisque les Rooms possèdent maintenant un attribut image, et cela a été compris dans le constructeur de Room. Lors de la création d'une Room on doit donc mettre une String de description et un chemin vers une image pour qu'elle s'affiche lorsqu'on pénètre dans cette Room.

De son côté, UserInterface permet de gérer les panels et l'emplacement donc de l'image, du champ de texte défilant(ou non défilant) du jeu (nommé « log » par défaut) et le champ d'entrée texte (là où le joueur doit taper les commandes). Et aussi, c'est là qu'on pourra implémenter un bouton.(voir exo bouton).

Exercice 7.18.7 optionnel: décrire addActionListener et actionPerformed :

addActionListener est une méthode qui permet d'ajouter un « listener », une interface qui permet de réagir (via des méthodes pour se déplacer par exemples) suite à un événement survenu dans le jeu.

La méthode actionPerformed permet de signaler, de déclarer au programmeur qu'une action (permise par l'interface listener associée) a été effectuée.

Exemple : je passe ma souris dans une certaine zone donc cela implique le déclenchement d'une commande.

Exercice 7.18.8 : ajouter un bouton

importer :

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import java.net.URL;

import java.awt.image.*;

Pour ajouter un bouton :

Pour ajouter un bouton, on crée 1 attributs : private JPanel pan; qui sera l'emplacement des boutons.

private JButton boutonSortie; qui est le bouton qui actionnera la commande « go sortie ».

Puis dans createGUI() :

pan = new JPanel();//declaration du panel pour les boutons

boutonSortie = new JButton("Sortie");//declaration du bouton

pan.add(boutonSortie);//ajout du bouton au panel

myFrame.getContentPane().add(panel, BorderLayout.CENTER);//positionnement de chaque panel

myFrame.getContentPane().add(pan, BorderLayout.SOUTH);

Résultat : Un bouton affichant « sortie » apparaît en bas de l'écran, on peut cliquer dessus, mais rien ne se passe.

Pour que ces boutons marchent :

dans createGUI :

boutonLook.addActionListener(this);//on pose un listener au bouton

et dans actionPerformed :

if(e.getSource()==boutonLook)//si on appuie sur le bouton

{

Engine.InterpretCommand("regarder");//le programme entre la ligne de commande "regarder" puis l'entre. Cela a le même effet qui nous tapions nous-même « regarder » dans le champ d'entrée texte.

}

La démarche a été effectuée pour toutes les commandes qui ne nécessitent pas de taper de second mot. Il a été jugé préférable de faire ainsi. Le but de la manœuvre est de rendre les commandes plus ergonomique, de gagner du temps. Or, taper un mot puis cliquer sur le bouton ne me paraît pas être un gain de temps, même plutôt une perte de temps. S'il faut taper quelque chose, autant tout taper.

Edit : en fin de projet, tous les boutons ont été testés, et fonctionnent. Le nom de chaque bouton a été traduit en français comme demandé par les professeurs (FR XOR EN).

Exercice 7.19 (OPTIONNEL)

Find out what the Model-View-Controller pattern is. You can do a web search to get information, or you can use any other sources you find.
How is it related to the topic discussed here ?
What does it suggest ?
How could it be applied to this project ? (Only discuss its application to this project, as an actual implementation would be an advanced challenge exercise.)

D'après le site http://baptiste-wicht.developpez.com/tutoriels/conception/mvc/ :

« Le design pattern Modèle-Vue-Contrôleur (MVC) est un pattern architectural qui sépare les données (le modèle), l'interface homme-machine (la vue) et la logique de contrôle (le contrôleur).

Ce modèle de conception impose donc une séparation en 3 couches :

- *Le modèle : Il représente les données de l'application. Il définit aussi l'interaction avec la base de données et le traitement de ces données.*
- *La vue : Elle représente l'interface utilisateur, ce avec quoi il interagit. Elle n'effectue aucun traitement, elle se contente simplement d'afficher les données que lui fournit le modèle. Il peut tout à fait y avoir plusieurs vues qui présentent les données d'un même modèle.*
- *Le contrôleur : Il gère l'interface entre le modèle et le client. Il va interpréter la requête de ce dernier pour lui envoyer la vue correspondante. Il effectue la synchronisation entre le modèle et les vues.*

La synchronisation entre la vue et le modèle se passe avec le pattern Observer. Il permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour. »

Il serait donc possible d'intégrer un nouveau genre d'interface graphique au jeu avec une partie uniquement dédiée aux mécaniques de déroulement du jeu, un mécanique graphique, et un interface de réception et traitement des commandes extérieures (celles du joueur).

Exercice 7.19.2

Déplacer toutes les images dans un répertoire Images à créer à la racine du projet.
Incorporer dans le jeu une image différente pour chaque Room.
S'il en manque, fabriquer une "image de mot" pour afficher simplement le nom du lieu

Les images ont été trouvées et enregistrées grâce à Google image, dans un dossier Images, à la racine de notre projet Zuul. Il y en a une pour chaque Room sauf les trois dernières, et celles-ci ont été redimensionnées à plusieurs reprises pour entrer dans le cadre de l'écran (redimensionnement effectué sur Paint). Etrangement, même si elles ont toutes la même dimension, elles apparaissent différemment à l'écran et la fenêtre s'agrandit/se redimensionne toute seule ce qui gêne le joueur.

Edit : une solution plus agréable pour le joueur a été trouvée en empêchant la fenêtre de se redimensionner trop et cacher l'entrée texte avec la méthode setResizable() appliquée à la JFrame. Et chaque Room a son image.

Exercice 7.20 et 7,21 et 7,22 :

Extend your adventure project so that a Room can contain a single Item.
Items have a description and a weight.
When creating Rooms and setting their exits, Items for this game should also be created.
When a player enters a Room, information about an Item present in this Room should be displayed.

Remarque : *il est possible de remplacer le poids de chaque Item par un prix, ou même de prévoir les deux !*

Modify the project so that a Room can hold any number of Items. Use a collection to do this.
Make sure the Room has an addItem method that places an Item into the Room.
Make sure all Items get shown when the player enters a Room.

Création d'une classe nommée « Item », avec deux attributs : un pour la description de l'objet, et un pour son poids. Dans cette classe, on réalise un constructeur qui fabrique un objet, définit par sa description et son poids entre autre.

Ajout dans Room, dans le constructeur d'une liste d'objets (HashMap) pour pouvoir ensuite créer ces objets dans la classe GameEngine, dans la méthode createRooms : on peut ajouter autant d'objet que l'on veut, de la même façon que l'on peut ajouter des sorties aux rooms.

(room.addItem(new Item(description, weight)) permet de créer un objet et le placer dans une room) avec la méthode addItem créer dans room pour la création d'objet comme ci-avant.

Edit : des modifications ont été apportées au sein des méthodes ainsi que leur place, notamment avec la création de la classe ItemList plus loin. Chaque Item possède désormais les attributs description(celle de l'objet), poids(poids de l'objet qui se répercute sur l'inventaire du joueur et sa capacité à le prendre ou non), transportable(pour savoir si le joueur a le droit de la prendre ou non), consommable(pour savoir si le joueur a le droit de la manger ou non), prix(certains objets peuvent se récupérer simplement, d'autres doivent être achetés).

How should all the information about an Item present in a Room be produced ?
Which class should produce the String describing the Item ?
Which class should display it ? Why ? Explain in writing.
If answering this exercise makes you feel you should change your implementation, go ahead and make the changes.

Ce procédé permet de créer un objet, et lui affecter toutes sortes d'informations sous forme de String en redéfinissant la méthode toString dans la classe item : description, poids, prix, dimensions,... La méthode toString redéfinie pour les objets se trouve naturellement dans la classe Item pour plusieurs raisons : les attributs y sont facilement accessibles, cela ordonne le projet, et cela diminue le couplage puisque des futures modifications, si modifications il y a besoin de faire, ne se feront que dans la classe Item.

Exercice 7,22,1 : optionnel

Justifier par écrit le choix du type de collection utilisé à l'exercice 7.22.

La collection que nous avons utilisée est une HashMap qui convient pour cet exercice puisqu'elle permet d'ajouter autant d'objet que l'on veut en se redimensionnant à chaque entrée d'objet dans la liste. On peut donc ajouter autant d'objet que l'on veut juste en les créant et on peut les retrouver directement grâce à la méthode get() de la HashMap et au jeu de clé/valeur les valeurs étant les objets, et les clés une chaîne de caractère à leur nom.

Exercice 7,22,2 :

Intégrer les objets (Items) de son jeu (au moins ceux du sous-scénario).

Les objets nécessaires au jeu ont été ajoutés. Il pourra être possible d'en rajouter si besoin est grâce aux propriétés de la HashMap. Au moins un objet par salle a été ajouté. Certains sont transportables, d'autres consommables, d'autres payants. Différentes interactions avec ces objets sont possibles, et ont été testé manuellement en jouant au jeu.

Edit : Tous les objets nécessaires pour le scénario sont présents dans chaque Room où cela est nécessaire en fin de projet.

Exercice 7.23 : back() :Commande qui permet de faire marche arrière :

Dans un premier temps, nous avons réalisé une commande back() qui permettait de retourner dans la dernière salle visitée. On débutait la partie avec comme attribut previousRoom=null et aCurrentRoom avec la premiere salle du jeu. A chaque changement de salle on opérait un changement : previousRoom=CurrentRoom puis CurrentRoom=NextRoom. Mais ce changement n'était pas viable, il ne fonctionnait pas toujours et en particulier lorsqu'on utilisait plusieurs fois back() en jeu : soit ça ne faisait rien, soit on se retrouvait bloqué dans une salle où aucune sortie ne fonctionnait. On a donc demandé de l'aide à A.LE (El groupe 5 projet zuul 2012/2013) qui était un peu en avance sur nous et qui nous a montré comment se servir d'une stack comme suit :

Dans GameEngine

Attribut: private Stack<Room> pileBack = new Stack<Room>();

puis dans goRoom on remplit la stackList :pileBack.push(currentRoom);

enterRoom(nextRoom);

puis dans la méthode back(), on utilise cette collection avec des conditions au cas où la pile est vide au moment où on veut l'utiliser, ou si back est suivi d'un second mot. Sinon on range le haut de la pile dans previousRoom, la dernière salle visitée.

Nous avons donc abandonné le précédent modèle pour celui-ci qui donne, en fin de projet le code suivant traduit en retour() :

```
/**
  * Retourne dans la dernière pièce visitee.
  */
 private void retour(Command command)
 {
   if (pileBack.empty())//si la stack est vide, et donc qu'il n'y a pas de room
precedentes
   {
     gui.println("Vous ne pouvez pas revenir en arrière !\n");
   }

   else if (command.hasSecondWord())//si le joueur ajoute qqchose apres le mot
commande
   {
     gui.println("Tapez juste retour pour revenir en arriere !\n");
   }

   else
   {
     Room previousRoom = pileBack.pop();//on recupere le top de la stack
     aPlayer.enterRoom(previousRoom);//on va dans cette room recuperee

     gui.println(aPlayer.getPlayerCurrentRoom().getLongDescription());//on
affiche le changement de zone

     if(aPlayer.getPlayerCurrentRoom().getImageName() != null)
     {
       gui.showImage(aPlayer.getPlayerCurrentRoom().getImageName());
//on affiche l'image de la zone
     }
 }  }
```

Exercice 7.24

Test your new command properly. Do not forget negative testing !
What does your program do if a player types a second word after the back command ?
Does it behave sensibly ? Are there more cases of negative testing ?

Les commandes fonctionnent (back avec second mot, back back et autres combinaisons testées).

Exercice 7.25

What does your program do if you type back twice ? Is this behavior sensible ?

Utiliser deux fois la commande back permet de revenir deux salles en arrière, comme attendu.

Edit : en fin de programme, la commande retour() (anciennement back()) peut être utilisé autant de fois que voulu jusqu'à remonter à la première Room. A ce moment, elle affiche qu'il est impossible de faire demi-tour. La commande retour() fonctionne aussi bien avec des Rooms qu'avec des TransporterRooms, ou avec l'utilisation d'un Beamer(téléporteur).

Exercice 7.26

Implement the back command so that using it repeatedly takes you back several rooms, all the way to the beginning of the game if used often enough. Use a Stack to do this. (You may need to find out about stacks. Look at the Java library documentation.)

La commande back fonctionne-t-elle si l'on est revenu au point de départ du jeu ? Modifier son jeu s'il n'a pas le comportement désiré.

Voir les 3 précendents exercices sur back()/retour().

Exercice 7.27 et 28.1 et 2 et3 :

What sort of baseline functionality tests might we wish to establish on the current version of the game ?

Nous avons dans un premier temps rempli un fichier texte (.txt avec le bloc-notes windows) dans lequel nous avons mis une unique commande de test pour voir si la fonctionnalité opérait (juste go nord).

La fonctionnalité de lecture de fichier est permise par l'utilisation de la classe Scanner comme suit :

```
/**
     * Methode de test des commandes via fichier texte.
     * On accede a un fichier texte pour en lire les instructions
     * automatiquement et verifiez que les commandes fonctionnent.
     */
    private void test(Command command)
    {

       if(!command.hasSecondWord())//s'il n'y a pas de 2nd mot
      {
         gui.println("Veuillez ajouter le nom du fichier de test à la suite de la
commande.");
      }

      else
      {
         String fichier = command.getSecondWord();

Scanner sc = new Scanner(
this.getClass().getClassLoader().getResourceAsStream("./"+fichier+".txt") ); //on
va lire le fichier issu du chemin

         String str = sc.nextLine();//lecture
         while(sc.hasNextLine())//tant que le fichier n'est pas fini
            {
               interpretCommand(str);//rentrer la commande str, ex str=go nord.
               str = sc.nextLine();//ligne suivante, la commande change.
            }                    }     }
```

En pratique dans le jeu, il faut taper test suivi du nom du fichier désiré. Si l'utilisateur ne tape pas le nom du fichier à la suite, le message d'erreur prévu dans la méthode apparaît.

Edit : Pour l'exercice 28.3, nous essaieront à l'avenir de mettre ce genre de phrases pour vérifier le bon fonctionnement du jeu.

Exercice 7.29 (Player) :

Cet exercice consiste à créer une classe nommée Player qui doit rapatrier tous ce qui concerne le joueur.

Nous avons donc crée cette classe avec des attributs propres au joueur : aCurrentRoom est la room dans laquelle se trouve le joueur, aName pour son nom, aArgent pour la somme d'argent avec laquelle le joueur commence le jeu, aPoidsMax le poids maximale que le joueur peut porter, aPoidsCourant le poids total des objets portés.

Il y a aussi un constructeur pour initialiser ces attributs, une méthode pour accéder à la room, une autre pour en changer. Ces méthodes étaient auparavant dans GameEngine.

Edit : en fin de projet, le joueur possède plus d'attributs et des accesseurs pour chacun d'entre eux.

Exercices 7.30 (take, drop) et 7.31 (porter plusieurs items) :

Les méthodes take et drop ont été réalisées dans gameEngine, elles permettent de mettre des objets de la Room dans l'inventaire du joueur. L'objet n'est plus dans room lorsqu'il va dans l'inventaire, et vice versa. Il est possible de porter plusieurs items puisque le joueur possède un inventaire qui est une itemList, elle-même est une hashMap.

Edit : en fin de projet, ramasser certains objets ont une influence sur le joueur (ex : le portefeuille augmente l'attribut argent du joueur), ou alors il est nécessaire que le joueur prenne certains objets pour interagir avec eux (ex : le téléporteur qui doit être dans l'inventaire du joueur, donc qui a été pris par le joueur, pour pouvoir actionner la commande charge du téléporteur et la téléportation ensuite, voir exercice du Beamer pour plus de détails à ce sujet). Voici les codes pour chaque commande :

```
/**
  * Propose de prendre l'objet qui est dans la zone courante.
  * Cela est possible si l'objet existe, s'il est dans la zone courante,
  * s'il est ramassable (voir attribut de l'item), si son poids
  * ne fait pas dépasser le poids max du joueur, si le joueur
  * a la somme d'argent necessaire pour l'acheter. Si oui, l'objet va
  * dans l'inventaire du joueur.
  */
  public void takeItem(Command command)
  {
    String itemString = command.getSecondWord();
    Item vItem = 
aPlayer.getPlayerCurrentRoom().getItemListRoom().getItem(itemString);


if(!command.hasSecondWord())//si pas de 2nd mot
       {
          gui.println("Que voulez-vous prendre ? \n");
          return;
       }


    else if 
(!aPlayer.getPlayerCurrentRoom().getItemListRoom().containItem(itemString))//
verifie si l'objet est dans la room
    {
          gui.println("Cet objet n'est pas dans la salle ! \n");
          return;
     }
```

```
else if(aPlayer.getArgent()<vItem.getPrixItem())//si l'objet est associe a un prix et que le joueur n'a pas les moyens.
    {
        gui.println("Vous n'avez pas les moyens pour ce genre folie... Cet objet coute trop cher.");
        return;
    }
else
    {
        if(vItem.testTransportable()==false || (aPlayer.getPoidsCourant()+vItem.getWeightItem())>aPlayer.getPoidsMax())
        {
            gui.println("Vous ne pouvez pas prendre cet objet, il est trop lourd pour vous !");
            return;
        }
        //on a verifie si l'objet etait transportable et si son poids ne fait pas depasser la limite imposer par aPlayer.aPoidsMax
        /*sinon:*/else {gui.println("Vous deposez dans votre inventaire : " + itemString + "\n");

aPlayer.setPoidsCourant(aPlayer.getPoidsCourant()+vItem.getWeightItem());//on modifie le poids courant du joueur avec l'arrivee de l'objet

aPlayer.getPlayerCurrentRoom().getItemListRoom().removeItem(itemString); //on enleve l'objet de la room courante pour ne pas le dupliquer
            aPlayer.getInventaire().addItem(itemString, vItem);//on ajoute l'objet a l'inventaire
            if(vItem.getPrixItem()>0)//si l'objet est payant
            {
                //on adapte le portefeuille du joueur.
                double vDifference=aPlayer.getArgent()-vItem.getPrixItem();
                aPlayer.setArgent(vDifference);
                gui.println("Vous avez depense "+vItem.getPrixItem()+" pour obtenir cet objet, et il vous reste "+vDifference+" en poche.");
            }
```

```
        }

        if(itemString.equals("portefeuille"))//si l'objet est le portefeuille
        {
          aPlayer.setArgent(20.0);//argent du joueur+=20 unités.
          gui.println("Vous avez ramasse votre portefeuille, il contient un billet de vingt");}}}
```

Et pour enlever l’objet de l’inventaire :

```
/**
   * pour lâcher un item, l'enlever de l'inventaire et le deposer dans la zone courante.
   */
  public void dropItem(Command command)
  {
     String itemString = command.getSecondWord();
     Item vItem = aPlayer.getInventaire().getItem(itemString);

     if(!command.hasSecondWord())//pas de second mot
       {
         gui.println("Que voulez-vous deposer ? \n");
       }
     else if(!aPlayer.containItemInventaire(itemString))//pas dans l'inventaire
     {
        gui.println("Cet objet n'est pas dans votre inventaire !");
     }
     else
     {
      gui.println("Vous deposez par terre : " + itemString + "\n");
      aPlayer.getPlayerCurrentRoom().getItemListRoom().addItem(itemString, vItem);//on ajoute l'objet a la room courante
```

```
        aPlayer.getInventaire().removeItem(itemString);//on enleve l'objet de
l'inventaire.
      }
   }
```

Edit : il a été rajouté un cas particulier pour la méthode qui permet de lâcher des objets à cause du portefeuille. En effet ce code permet d'avoir de l'argent de manière illimité en lâchant le portefeuille et en le reprenant. Il a été corrigé en interdisant au joueur de pouvoir le reposer.

Evidemment, la démarche pour ajouter une commande a été opérée sur takeItem() et dropItem(voir la structure en switch et les CommandWord).

<u>Exercice 7.31.1 : ItemList :</u>

La classe ItemList a été créée. Elle permet de construire des listes (HashMap<String,Item>) d'objets, avec des méthodes associées : pour ajouter des objets, en enlever, vérifier s'ils y sont etc.

<u>Exercice 7.32 : poids max :</u>

La classe Player possède un attribut aPoidsMax qui fixe le poids maximal que le joueur peut porter : il est comparé au poids de tous les objets de l'inventaire et si un objet veut être ajouté à l'inventaire (voir commande exercice take/ méthode takeItem(), ajouter son poids ne doit pas faire dépasser la limite. Sinon un message d'erreur apparaît et l'objet reste dans la room.

Dans le registre des restrictions, la notion de prix a son importance aussi puisque si le joueur n'a pas assez d'argent sur lui, il ne peut pas acheter certains objets payants.

Exercice 7.33 : afficher l' inventaire du joueur :

La commande « objets » permet, une fois qu'elle est tapée, d'afficher à l'écran le contenu de l'inventaire et les caractéristiques de chaque objet (description, prix, poids…).

Exercice 7.34 : magic cookie :

Un objet « cookie magique » a été créé et placé dans une room. Une fois qu'il a été placé dans l'inventaire, le poids maximal que le joueur peut porter (aPoidsMax) est modifié (via un modificateur) et il augmente de 50%.

De la même manière dans le jeu, boire de l'alcool est punitif et sa consommation fait diminuer cet attribut de 20%.

Exercices 7.34.1 et 2 :

Le fichier de test n'est pas totalement mis à jour et les javadoc sont regénérées, mais les commentaires sont aussi à mettre à jour/ à traduire.

Edit : les javadocs et leurs commentaires sont à jour en fin de projet. Les fichiers tests aussi.

Exercice 7.35 : zuul with enums v1 :

Projet étudié et modifications au projet apportées.

Exercice 7.35.1: switch :

La structure Switch a été apportée pour la reconnaissance des commandes tapées, comme dans le projet zuul-with-enums-v1. La voici en fin de projet:

```
switch ( commandWord ){

    case HELP: printHelp();

    break;

    case GO: goRoom(command);

    break;

    case BACK: retour(command);

    break;

    case REGARDER: look();

    break;

    case MANGER: eat(command);

    break;

    case TEST: test(command);

    break;

    case PRENDRE: takeItem(command);

    break;

    case LACHER: dropItem(command);

    break;

    case RESTART: restart();

    break;

    case OBJETS: objets();

    break;

    case CHARGE: charge(command, aPlayer.getPlayerCurrentRoom());

    break;
```

```
case TELEPORTATION: teleportation(command);
break;
case SAVE: save(command);
break;
case LOAD: load(command);
break;
case QUIT:        if(command.hasSecondWord())
  gui.println("Tapez seulement 'quit' pour mettre un terme à la partie.");
else
  endGame();
break;
```

Exercice 7.37 et 38 et 41: translate & help() :

La commande help a été traduite en « aide »dans la classe «CommandWords » et le changement s'est opéré partout, à l'exception de la méthode printLocationInfo puisque help était une chaîne de caractère. Il a juste fallu la remplacer par CommandWord.HELP.toString().

Edit : cette modification a été aussi apportée au bouton qui voient donc leur nom traduit lors du jeu en français.

Exercice 7.40 : zuul with enums v2 :

Projet étudié et modificatiosn apportées.

Exercice 7.41.2 : nouvelle IHM :

L'équipe ne désire pas de nouvelle interface donc l'exercice ne sera pas fait.

Exercice 7.42 et 42.1: Time limit.

Hors du projet, nous avons testé une méthode qui permet d'afficher un message au bout d'un temps voulu en utilisant le temps réel avec la classe « getCurrentTimeMillis ». Nous ne l'avons pas encore intégrer au projet, nous cherchons quel objectif pourrait être réalisé en temps limité. Cela sera fait d'ici peu.

Edit : voici la méthode qui limite le temps de jeu à 30 minutes :

```
/**
 * Determine le temps au bout duquel le joueur doit avoir termine sa partie.
 * Au-dela de ce temps, c'est un echec.
 */
public void tempsImparti()
{
    long Debut = System.currentTimeMillis();
    long tempsActuel=0;
    while( (tempsActuel-Debut) <1800000 )//tant que 1800000 millisecondes ne se sont pas écoulées depuis le lancement de la partie.
    {
        tempsActuel=System.currentTimeMillis();


        //if(tempsActuel-Debut = 600000){gui.println("Vite! Il vous reste 20 minutes !");}
    }
    engine.echecTime();
}
```

La méthode a été testée en laissant la console tourner le temps nécessaire.

Edit2 : en fin de projet, un problème s'est posé lors de l'exercice consistant à créer une applet. Cette méthode empêche le jeu de se lancer. L.DUPONT (E1 projet zuul 2012/2013) m'a expliqué que cette méthode qu'il a lui aussi réalisé avec getCurrentTimeMillis() empêche le jeu de se lancer tant que le chronomètre lancé dans la méthode n'est pas fini. Il m'a donc aidé à réaliser un Timer à partir du code d'un groupe d'étudiant ayant réalisé leur projet zuul l'année dernière à l'ESIEE : voir emprunt du code de Aymeric KOEPPEL, Sébastien HU, Frédéric NGUYEN, Justine BOUGNOL (étudiants à ESIEE Paris) : projet Zuul 2011/2012 "Death Ception" dans mes classes GameEngine, CountDown et UserInterface. Cela permet de lancer l'applet sur le site du jeu, et le chronomètre a une fois de plus été testé en laissant tourner la machine le temps nécessaire pour voir s'afficher le message de fin de jeu sans quitter la première pièce.

Exercice 7.43 : *trap door* :

La solution de facilité consiste à créer une direction permettant d'aller dans une room, mais de ne pas créer la direction inverse. (ex : room accessible en tapant go nord, mais impossibilité de faire go sud).

Il pourrait être plus pratique de créer une classe « porte » avec des portes ouvertes/fermées à clefs/infranchissable.

Edit : en fin de projet, la classe door n'a pas été réalisée puisque le système de « trap door » suffit pour empêcher le joueur de faire demi-tour lorsqu'il est sorti du bar, et qu'il doit faire son choix entre les deux chemins.

Exercice 7.44 : Beamer, réalisé avec l'aide de Léo Dupont, monôme 5L.

J'avais dans un premier temps créer simplement un item appelé « teleporteur » dans GameEngine avec deux méthodes (chargement() pour récupérer la room courante et teleportation() pour changer de salle) dédié uniquement à cet objet. Voici son code :

```
//avant la classe beamer
//      if(!aPlayer.containItemInventaire("teleporteur"))//cas où le teleporteur n'est pas dans l'inventaire
//         {
//            gui.println("Vous n'avez pas de teleporteur dans votre inventaire !");
//         }
//      else if(this.aVariable==null){gui.println("Le teleporteur n'est pas charge.");}
//
//      else{
//            aPlayer.enterRoom(aVariable);
//      if(aPlayer.getCurrentRoom().getImageName() != null)
//         {
//            gui.showImage(aPlayer.getCurrentRoom().getImageName());
//         }
//         gui.println("Vous venez de vous teleporter !");
//         aPlayer.getCurrentRoom().getLongDescription();
//         this.aVariable=null;
//      }
// echec
//      for(Map.Entry<String,Item> e : aPlayer.getInventaire2().entrySet())
//      {
//          if (e.getKey()!="teleporteur" )
//          {
//            gui.println("Vous n'avez pas de teleporteur dans votre inventaire !");
//          }
//          else if(getCharge()==false || getDestination()=null )
//          {
```

```
//            gui.println("Le teleporteur n'est pas charge !Il ne peut donc vous emmener nulle part ! ");
//          }
//          else{aPlayer.enterRoom(getDestination());
//             if(aPlayer.getCurrentRoom().getImageName() != null)
//             {
//                gui.showImage(aPlayer.getCurrentRoom().getImageName());
//               }
//               gui.println("Vous venez de vous teleporter !");
//               aPlayer.getCurrentRoom().getLongDescription();
//          }
//      }
```

Cette méthode fonctionnait ainsi que celle pour charger.

Mais Léo Dupont (étudiant dans mon groupe) m'a conseillé de créer une classe Beamer qui hérite de la classe Item avec ces méthodes pour pouvoir parer au cas où je voudrais créer plusieurs téléporteurs. Aussi les exercices suivant nous incitaient à le faire puisqu'il y est question d'héritage. Cela a donc donné lieu à une classe Beamer et aux méthodes suivantes :

Pour charger :

```
/**
    * permet de charger le teleporteur pour une future utilisation.
    * Il pourra nous amener instantanement la ou nous l'avons charge.
    * @param commande charge
    * @parame Room la zone ou on effectue la charge.
    */
    public void charge(final Command pCommande, final Room pDestination)
    {
//      //Identifier le teleporteur
       String vStringTele = pCommande.getSecondWord(); // pour utiliser charge sur tel teleporteur
//       //Mais il faut verifier si l'objet est un teleporteur
```

```
        Item vItemOrBeamer = this.aPlayer.getInventaire2().get(vStringTele);//on
recupere l'objet
        Beamer vTeleporteur = new Beamer();
         //Verifier s'il est dans l'inventaire
           if(vItemOrBeamer==null){gui.println("Vous n'avez pas ce teleporteur sur
vous !");return;}
    //        //Verifier si l'objet est un teleporteur
            try
         {
              //on tente de le caster
             vTeleporteur = (Beamer)vItemOrBeamer;
         }
         catch(final java.lang.ClassCastException pE)
         {
            gui.println("Vous ne pouvez pas charger cela, ce n'est pas un teleporteur !");
return;
        }

        //Maintenant on peut charger le teleporteur
        vTeleporteur.chargement(this.aPlayer.getPlayerCurrentRoom());
        gui.println("Ce teleporteur est charge");   }
```

Puis pour se téléporter :

```
/**
     * permet d'utiliser le teleporteur. Il nous amene la ou on l'a charge.
     * @param commande teleportation.
     */
    public void teleportation(final Command pCommande)
    {//methode actuelle
            String vStringTele = pCommande.getSecondWord(); // pour utiliser charge
sur tel teleporteur
//        //Mais il faut verifier si l'objet est un teleporteur
        Item vItemOrBeamer = this.aPlayer.getInventaire2().get(vStringTele);//on
recupere l'objet
        Beamer vTeleporteur = new Beamer();
```

```
if(vItemOrBeamer==null){gui.println("Vous n'avez pas ce teleporteur sur vous !");return;}

try
{
    //on tente de le caster
   vTeleporteur = (Beamer)vItemOrBeamer;
}
catch(final java.lang.ClassCastException pE)
{
   gui.println("Vous ne pouvez pas charger cela, ce n'est pas un teleporteur !");
return;
}

//On verifie maintenant que le teleporteur est charge
if(vTeleporteur.getCharge()==false)
{
   gui.println("Ce teleporteur n'est pas charge.");return;
}
else{
   gui.println("Vous vous etes teleporte dans une nouvelle piece !");
   aPlayer.enterRoom(vTeleporteur.getDestination());
   if(aPlayer.getPlayerCurrentRoom().getImageName() != null)
      {
         gui.showImage(aPlayer.getPlayerCurrentRoom().getImageName());
        }
        gui.println("Vous venez de vous teleporter !");
        aPlayer.getPlayerCurrentRoom().getLongDescription();
   }        }
```

Ces deux méthodes ont été réalisées avec l'aide de L.DUPONT (E1 groupe5).

Exercice 7.45 : Optionnel, la classe Door

Une classe Door a été commencée mais le peu de temps restant et le peu d'utilité pour mon jeu ne me permettent pas de la terminer. Dans le principe, il s'agirait de remplacer les sorties (« exits » dans le code initial zuul-bad) par des portes avec un attribut de type Room pour chaque côté de la porte (l'entrée c-a-d la room où le joueur se trouve, et la sortie donc la room où le joueur peut aller), un attribut booléen pour verrouiller la porte (elle est soit ouvrable, soit fermée à clef et il y aurait donc un attribut de type Item correspondant (une clef spécifique à la porte), soit fermée à clef mais sans clef associée donc impossible à ouvrir mais peut-être contournable par un autre chemin ou téléportation), et enfin un attribut booléen pour gérer ou non la fermeture automatique de la porte après le passage du joueur lui interdisant ainsi de rebrousser chemin.

Edit : en fin de projet, la classe Door a été abandonnée, mais le début de rédaction de cette classe est disponible dans le projet et dans la javadoc.

Exercice 7.46 : Transporter Room : (avec l'aide de Léo Dupont E1).

Il s'agit dans cette exercice de crée une nouvelle catégorie de Room, une Transporter Room, donc une Room qui lorsqu'on veut en sortir nous amène dans n'importe quelle pièce du jeu. Une classe TransporterRoom héritant de Room a donc été créée avec un attribut de type RoomRandomizer, ce genre d'objet ayant lui aussi sa classe dans le jeu et permettant de définir les Rooms dans lesquelles nous pourront nous rendre et d'en choisir une « au hasard ».

La classe RoomRandomizer a pour attribut une HashMap de toutes les rooms, HashMap de String et de Rooms remplie à partir de la classe Room, une Arraylist dans laquelle on y met juste les Rooms et d'une méthode de séléction qui avec un objet de type Random apporte un nombre « aléatoire » compris en 0 et le nombre de room dans la liste, et la méthode retourne la Room correspondant à l'indice de la liste égale au nombre aléatoire. Finallement, on se sert de cette Room dans la classe TransporterRoom en redéfinissant la méthode getExit() pour apporter la nouvelle direction/room.

Ces classes TransporterRoom et RoomRandomizer ont été achevées et débuggées avec l'aide de L.DUPONT (E1 groupe5).

Edit : en fin de projet, le jeu comporte deux TransporterRoom, les deux sont fonctionnelles. L'une est disponible en début de jeu (cabine des toilettes), l'autre en milieu de jeu (cabine téléphonique).

Exercice 7.46.1 : commande alea()

Consigne floue, non comprise.

Exercice 7.46.2 : amélioration par l'héritage des exercices 43 à 45

La classe Beamer a été créée dans cette optique, elle découle (« extends ») de la classe Item, puisque le téléporteur est type d'objet. La première version de notre téléporteur était simplement un objet qui déclenchait une téléportation sous certaines conditions (si l'objet a pour String associée les caractères formant «teleporteur » , s'il est dans l'inventaire, etc). La classe Beamer a été réalisée pour cet exercice bien qu'elle ne serve pas vraiment pour le scenario du jeu (époque contemporaine). La classe Door a été commencée mais non finie car elle impliquait des changements inutiles au jeu, une simple « trap door » étant suffisante.

Exercice 7.47.3 : commentaires javadoc du jeu :

Un commentaire en français accompagne chaque méthode comme ci-après :

```
/**
*Commentaires
*(Version + date de création ou dernière modification signalée dans le
*commentaire de classe).
*@return et/ou @param
*/

Public/private type m()

{

.../*---*/... ;   //commentaire supplémentaire si besoin.

}
```

Les emprunts de codes ne sont effectués que si le code en question est compris et ils sont signalés dans le commentaire de la méthode et/ou de la classe si le code est éparpillé et dans le rapport pour l'exercice en question en tant que source si le code est copié, ou aide/collaboration si le code emprunté possède des modifications ou n'a servi que de base pour réaliser l'exercice.

Exercice 7.47 : abstract command/zuul-even-better

Nous avons tenté de faire cet exercice. Tout compilait après création de la classe abstraite dédiée aux commandes et des classes GoCommand et QuitCommand sans avoir rempli la méthode execute(). Nous avions même réalisé une classe pour chacune des commandes (autour d'une quinzaine). Mais le soucis est que nous n'arrivions pas a utilisé la méthode println() de UserInterface. En effet nous n'arrivons à nous en servir que dans la classe GameEngine, et comme toute commande s'accompagne forcément d'un affichage de caractères dans le jeu nous avons dû abandonner l'exercice malgré ses avantages intéressants en matière de couplage puisqu'il désengorgeait grandement la classe GameEngine qui fait office désormais de « fourre-tout » malheureusement.

Exercice 7.47.1 : les paquetages

Nous avons créé deux paquetages : l'un nommé pkg_jeu qui regroupe les classes liées au joueur et à son avancée (Player, GameEngine, Item, ItemList, Beamer, Room, TransporterRoom, RoomRandomizer), et l'autre pkg_structure qui renferme les classes liées à la mécanique du jeu (UserInterface, Parser, Command, CommandWord, CommandWords).

Comme demandé, la classe Game est en dehors de ces paquetages, les noms commencent par pkg_, il y en a (au moins) 2, et ils contiennent plus de 2 classes chacun.

Exercice 7.48 : Character

La classe Character a été créée dans le pkg_jeu. Elle crée des personnages caractérisés par un nom, la room courante (dans laquelle il se trouve), une String qui est une phrase qui s'affichera lorsqu'il rencontrera le joueur. Les personnages sont créés dans la classe GameEngine, au moment de créer les rooms et ils sont directement placés dans une HashMap<String,Character> prévue à cet effet pour y accéder facilement plus tard si besoin. Aussi, une méthode getListeCharac() a été réalisée dans Room pour afficher les joueurs présents dans la room ou se situe le joueur.
En marge de cet exercice, une méthode pour afficher les dialogues de chaque joueur a été créée ainsi qu'une méthode qui regroupe les interactions contextuelles entre le joueur et le personnage (i.e. la réaction de tel personnage en fonction de certains attributs ou possession du joueur : le policier est sensible au fait que le joueur puisse avoir de l'alcool ou de la drogue sur lui et peut le punir, la femme du joueur est sensible au fait qu'il soit saoul ou lucide, etc…)

Exercice 7.49 : MovingCharacter

Une méthode movingCharacter() a été produite dans GameEngine pour déplacer un personnage selon certaines conditions. Le personnage est au comptoir du bar initialement. Si le joueur pénètre dans la grande salle, alors le personnage de déplace vers les tables des clients, et y reste. Puis, si le joueur va au comptoir, alors dans le même temps le personnage vient au comptoir pour discuter avec le joueur.

Exercice 7.49.2 : incorporez les éléments du scenario

Les items et personnages manquants pour le scenario ont été ajouté au jeu. Un des personnages se déplace pendant la progression du joueur, tous les lieux sont accessibles et tous les objets récupérables/consommables/payants le sont. Les personnages prononcent une phrase lorsqu'ils croisent le joueur, et certains événements se déclenchent avec eux si certaines conditions sont remplies (être saoul ou lucide, avoir certains objets…). Ils nous donnent des informations ou des objets clé pour progresser ou pour faire face à la femme du joueur qui fait office d'ennemi vous attendant au tournant en fin de jeu.

Exercice 7.53 : méthode main()

La méthode main() a été ajoutée dans la classe Game.

```
/**
 * Permet de lancer le jeu.
 */
public static void main( String[] pArgs )
{
Game game = new Game();
}
```

Exercice 7.54 : sans blueJ

On peut lancer le jeu avec la méthode main() ci-dessus en y accédant depuis la panneau de commande de l'ordinateur en y tapant :

CD
chemin vers le jeu

Ou en créant un fichier .jar dans blueJ avec pour classe principale Game, on peut lancer le jeu par un simple double-clic ou dans le panneau de commande en plaçant le chamin vers le fichier.jar .

Exercice 58 : fichier .jar

Un fichier .jar avec pour classe principale Game a été créé avec BlueJ.

Exercice 58.1 : lancer le jeu avec la commande java – jar

Voir exo 54.

Exercice 58.2 : Pas de IHM prévue.

Exercice 59 : jeu en applet

Pour pouvoir faire de notre jeu une applet,

dans Game :

public class Game extends javax.swing.JApplet {//contenu de la classe Game}

Puis on peut ajouter dans UserInterface des méthodes pour gérer l'affichage et le g.u.i. pour empêcher la machine de redimensionner l'image trop souvent ou fermer la fenêtre de jeu une fois le jeu terminé, etc..

Exercice 59.1,60, 60.1, 60.2, 60.3, 60.4 : applet en ligne, IHM, javadoc.

Il faut placer le code suivant dans le fichier index.html de la page web où l'on veut que l'applet apparaisse:

```
<BODY>
    <applet archive= "chemin vers le fichier .jar sur le serveur"
        code="Game"
        alt="Votre navigateur ne supporte pas Java"
        name="Epopee ethylique a Dublin">
    </applet>
</BODY>
```

Une icône java nous propose alors de lancer l'application pour jouer. Je l'ai testé sur plusieurs ordinateurs pour vérifier son accessibilité. Aucune IHM graphique n'était prévue. La javadoc est complétée au mieux, en français, et elle est consultable sur le site.

Exercice 61 : Sauvegarde du jeu, 62 :Chargement du jeu, 62.1 : tests de sauvegarde/chargement :

Avec l'aide de L.DUPONT(E1, groupe5) nous avons essayé de nous lancer dans cet exercice. Nous avons créé deux méthodes, une pour sauvegarder et l'autre pour charger la progression du joueur. La méthode save() permet de créer un fichier .txt dans lequel nous y mettons des informations, des attributs du joueur ou de la Room grâce notamment à une redéfinition de toString() dans la classe Player et grâce aussi à la classe PrintWriter. Donc en tapant dans le jeu save fichierA, nous créons dans le dossier du projet un fichier nommé fichierA.txt qui contient les informations détaillées ci-avant. Puis en tapant load fichierA dans le jeu, nous utilisons la classe Scanner pour lire le fichier nommé fichierA et utilisons certains méthodes pour appliquer des modificateurs d'attributs.

Voici la méthode save() dans GameEngine :

```
  /**
   * Pour sauvegarder la progression du joueur.
   * On cree un fichier texte avec l'etat du joueur au moment ou on rentre la commande.
   */
  public void save(Command command)
  {
    // === Dans quel fichier === //
    if (!command.hasSecondWord()) {
      gui.println("Veuillez spécifier un nom de sauvegarde.");
      return;
    }
    String vNomFichier = command.getSecondWord();
    /* création du fichier de sauvegarde */
    PrintWriter vPW;
```

```
    try {vPW = new PrintWriter("./saves/" + vNomFichier + ".txt");}
    catch (final IOException pExp) {
        gui.println("Impossible de créer le fichier.");
        return;
    }


    // === Ecritures dans le fichier === //
    /* sauvegarde des attributs de Player */
    vPW.println((aPlayer.toString()));


    // === Fermeture du fichier === //
    vPW.close();
}
```

Et voici la méthode load() dans GameEngine :

```
/**
  * Pour charger une partie deja sauvegardee.
  * On rentre un nom de fichier avec l'etat du joueur
  * au moment ou il a sauvegarde sa progression.
  * Le jeu reprend avec les nouveaux attributs issus du fichier .txt .
  */
public void load(Command command)
{
    // === Quel fichier === //
    if (!command.hasSecondWord()) {
        gui.println("Veuillez spécifier un nom de sauvegarde.");
        return;
    }
```

```
        String vNomFichier = command.getSecondWord();
        /* création du fichier de sauvegarde */
        Scanner vSc;
        try {vSc = new Scanner(new File("./saves/" + vNomFichier + ".txt"));}
        catch (final java.io.FileNotFoundException pExp) {
            gui.println("Sauvegarde introuvable.");
            return;
        }
        vSc.useDelimiter(":");

        // === Lecture du fichier === //
        while (vSc.hasNext()) {
            String vS = vSc.next();
            /* attributs de Player */
            if (vS.equals("\n")) {}
            else if (vS.equals("poids")) {
                int vPoids = vSc.nextInt();
                aPlayer.setPoidsMax(vPoids);
                gui.println("Le nouveau poids max du joueur est: "+vPoids);
            }
        }
        // === Fermeture du fichier === //
        vSc.close();
    }
```

Ces méthodes ont été testées pendant le jeu avec uniquement l'attribut aPoidsMax du joueur qui peut être modifié dans certains phase de jeu, notamment en buvant de l'eau (l'attribut augmente de 20%) et cela fonctionne.

Edit : en fin de projet je n'ai plus assez de temps pour gérer la sauvegarde des autres attributs avec la méthode toString() dans Player que voici :

```
@Override public String toString()
  {
     return "poids:"+this.aPoidsMax+""+":";
  }
```

Qui donne donc dans le fichier texte :

Poids : 108 :

Avec « : » pour délimiter les données.

Aussi, L.Dupont a continué cet exercice de sauvegarde et a réussi à sauvegarder toutes les données possibles (tous les attributs de son joueur, la room courante au moment de la sauvegarde, les rooms précédement visitées pour pouvoir utiliser la commande back()/retour(), les actions de son joueur, l'inventaire du joueur, le temps écoulé, le contenu des rooms(objets et personnages), le log(le champ texte défilant où se déroule le jeu), tout ce qu'il faut pour reprendre la partie là où on l'a arrêtée en somme.

De mon côté j'ai testé cette sauvegarde à différents endroits du jeu et avec différents poids max portables (puisque je n'ai sauvegardé que cela) et le jeu fonctionne parfaitement.

Exercice 63 : Scenarios de test :

Trois scenarios de tests ont été réalisés : un qui permet de gagner directement (rentrer : « test win_direct »), un qui explore toutes les pièces du jeu (« test toutes_pieces ») et un qui effectue chacune des commandes possibles du jeu (« test toutes_commandes »).

Ces scenarios ont permis de trouver beaucoup de failles dans le jeu, notamment avec les méthodes pour prendre des objets, les lâcher, les manger qui donnaient lieu à des incohérences (lâcher un objet sans que le poids courant porté par le joueur ne diminue, etc… Ces changements ont été apportés donc ils n'apparaissent pas dans le rapport. Néanmoins, vous les trouverez dans la java/prog/userdoc).

Exercice 63.1 : Compilation des paquetages :

Tout le projet compile et s'exécute comme voulu, y compris les paquetages.

Exercice 63.2 et 63.3 : Rendre le jeu jouable :

Le jeu est désormais jouable, c'est-à-dire que tous les éléments de scénarios sont présents, comme tous les objets et tous les personnages. Il est possible de gagner ou de perdre le jeu selon les actions du joueur. Les conditions pour gagner le jeu sont expliquées sur le site web du jeu : http://esiee.fr/~houacinm/zuul/ parties scénario, scénario détaillé et situations gagnantes/perdantes.

Exercice 63.3 :Mise à jour finale du site+java/prog/userdoc :

Les documentations du projet sont à jour et disponibles comme précédemment sur le site du jeu : : http://esiee.fr/~houacinm/zuul/ en bas de page. Le site est quant à lui complet.

IV.B) Exercices en plus pour peaufiner le projet :

1. revérifier qu'il y a bien une image pour chaque 'Room'

2. homogénéiser les messages et dialogues (français XOR anglais)

3. commenter les sources (en entête: les auteurs + citations pour **éviter le plagiat** + courte description; pour chaque méthode: courte description + paramètres + return + throws; et quelques commentaires indispensables dispersés dans le code)

4. corriger l'orthographe et les fautes de frappe dans le jeu, dans les sources, et dans le rapport

5. regénérer la javadoc en utilisant les options -private et -linksource à partir de la ligne de commande

6. compléter le rapport avec entre autres table des matières, scénario, plan, explications sur les exercices et sur les bouts de code empruntés, scénarios de test, et récapitulatif des contributions de chacun.

7. vérifier la page web (titre, noms, rapport (ou scenario, plan, ...), applette, etc.)

8. préparer le dépôt JNews

9. se faire expliquer toutes les parties que l'on n'a pas directement programmées

10. le fichier README.TXT à la racine du répertoire contenant le projet (c'est-à-dire à la racine du .jar) doit comporter l'URL complète permettant d'accéder à la page web de l'équipe.

1/Le problème de la première Room du jeu persiste sans raison apparente. Pour rappel, leu jeu n'arrive pas à l'afficher au lancement, mais n'a pas de soucis pour l'afficher après quand on y revient.

2/Les commandes, commentaires des méthodes, documentations, dialogues du jeu, direction, nom des zones, nom des personnages, nom des objets, et autres indications dans le jeu sont exclusivement en français mise à part peut-être quelques noms de méthodes mais l'utilisateur n'aura affaire qu'à une interface totalement francophone.

3/Chaque classe possède ses commentaires en en-tête et dispersés dans le code si besoin pour certains méthodes (ex :beamer, transporterRoom,etc), son auteur, ses sources. Les emprunts de codes sont précisés pour éviter tout plagiat dans le code, et sur le site web.

4/Le jeu a été relu par son auteur (Mehdi HOUACINE), et testé par d'autres personnes du groupe 5 notamment au cas où il y aurait des fautes que je n'aurais pas vu où des incohérences dans le jeu en plus des fichiers de tests.

5/Les javadocs ont été générées grâce aux lignes de commandes fournies dans la liste officielle des exercices du projet zuul. Elles sont entreposées dans un fichier .bat pour pouvoir les re-générer rapidement.

6/Le rapport contient l'approche de l'auteur du jeu vis-à-vis de chaque exercice. J'y précise quand c'est le cas les contributions de personnes extérieures au « binôme » et les emprunts de code pour un exercice donné.

7/La page web contient tous les éléments nécessaires : le plan imposé pour le projet (voir première page du projet), l'applet pour y jouer, etc.

8/Tous les éléments pour déposer le projet sont prêts.

9/Tous les exercices y compris la totalité de la rédaction de ce rapport et du scénario et du site web ont été réalisées par Mehdi HOUACINE. Je n'ai pas eu de nouvelles de L.Defloris depuis un peu plus de deux mois (nous sommes le 12/05) et sa participation au projet auparavant est quasi-inexistante (je vous renvoie aux 3 premiers exercices de la liste). Il ne s'est jamais manifesté hors des premières séances de TP projet pour participer au développement du jeu « Epopée éthylique à Dublin » ou pour se faire expliquer les parties de code du projet malgré mes mails et mises en garde au début du projet (à peu près jusqu'à l'exercice 15).

10/Le fichier README.txt contient bien l'url du site web de présentation du jeu.

IV. C) Scenarii de test :

Ce premier scénario permet de gagner le jeu sans faire de détour. Il permet de savoir s'il est possible de gagner le jeu. Aucun soucis ne doit nous en empêcher :

prendre eau

manger eau

go sud

go sortie

go ouest

go ouest

go sud

go sud

go est

go entree

go sud

go etage

Ce second scenario permet de visiter toutes les zones du jeu. On vérifie donc si chaque zone est accessible et s'il est possible d'en sortir. C'est le cas puisqu'on arrive à la dernière salle en lançant ce scenario et qu'on a visité chaque pièce :

prendre eau

manger eau

regarder

go ouest

retour

go sud

prendre teleporteur

go est

go ouest

go etage

prendre portefeuille

go descendre

go sortie

charge teleporteur

go ouest

go ouest

go entree

go sortie

go sud

go entree

go sortie

go sud

go entrée

go est

teleportation teleporteur

regarder

go est

go est

go entree

go nord

retour

retour

go est

go entree

go sud

go entree

go sortie

go ouest

go entree

go sud

go etage

Ce scenario permet quant à lui de tester la validité de toutes les commandes de je. On vérifie donc que chaque commande produit l'effet voulu et que des fausses manipulations/commandes n'empêchent pas la progression ::

prendre cookie

manger cookie

prendre eau

go sud

lacher eau

go est

prendre table

lacher table

lacher lehvgheosuivnhiv

prendre alcool

retour

prendre salle

prendre teleporteur

manger teleporteur

go ouest

charge teleporteur

prendre transporteur

manger transporteur

retour

go etage

prendre billard

lacher teleporteur

prendre portefeuille

go descendre

go ouest

lacher transporteur

prendre eau

prendre cafe

manger cafe

manger alcool

regarder

manger eau

teleportation transporteur

retour

charge transporteur

go sortie

go est

retour

go ouest

go ouest

go entree

prendre cachet

retour

prendre cachet

manger cachet

teleportation transporteur

prendre comprime

manger comprime

go entree

prendre comprime

prendre comprime

manger comprime

go sud

go etage

prendre teleporteur

teleportation teleporteur

go est

charge teleporteur

go nord

go ouest

go

gogogogo

go droite

retour

go gauche

teleportation teleporteur
objets
regarder
aide
restart
save EpopeeTest
go sud
prendre eau
manger eau
objets
regarder
load EpopeeTest
regarder
quit